Unit 6. Geometrical Processes II: Transformations

6.1 Geometric Transformations

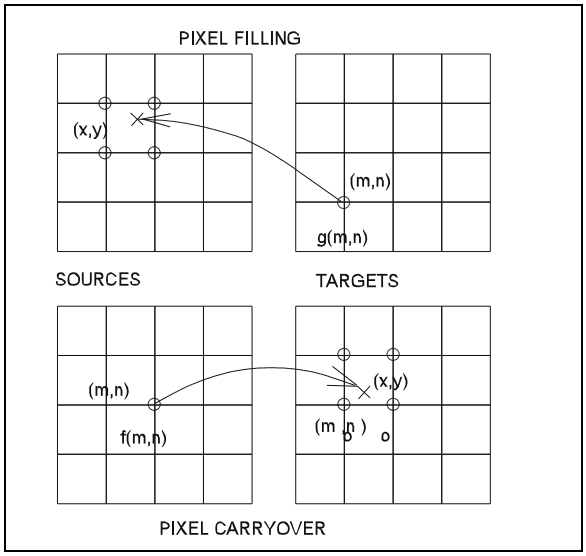
Pixel Mappings. A *geometrical* transformation of a *source* image {f(m,n)} into a *target* image {g(a,b)} moves the source pixel locations (m,n) to target locations (a,b) and assigns them gray levels. There may be fewer or more pixels in {g(a,b)} than in {f(m,n)} and so interpolation is necessary to obtain gray levels for them. Such mappings are necessary to map a distorted image in a manner that removes the distortion. Also, an image may not have the correct aspect ratio (height-to-width) and needs to be stretched in one direction or another. Other reasons exit, such as zooming in or out. If an image taken the previous year is to be subtracted from one taken this year, for example, to show the new growth or deforestation, at least one image must be *registered* with the other one (processed to "fit" the other one) whenever the images were not taken from the same point in space (and perhaps not with the same lens or angle or in the same band of light, infrared or radar or from the same distance).

The general formulation for a geometric transformation of *pixel locations* is

$$(m,n) \Rightarrow (a,b) = (a(m,n),b(m,n)) \tag{6.1}$$

where $\{f(m,n)\}$ is the input image, $\{g(a,b)\}$ is the output image, and $a = a(m,n)$ and $b = b(m,n)$ specify the transformation $(m,n) \Rightarrow (a,b)$. The transformations are continuous and although the arguments (m,n) are integer values, they can map to real nonintegral values (a,b).

Figure 6.1. Mapping pixels.



Because the pixel locations (a,b) in the new (target) image must be located at integer pixel values, they may map to nonintegral values (x,y), so interpolation is necessary to find the gray level at the integral point (a,b). The gray levels at the nonintegral points are used in this interpolation. Interpolation methods are discussed in a later section.

In Figure 6.1, the top mapping shows a pixel location (m,n) in the *target* (output) image with gray level $g(m,n)$. Going backwards, from target to *source* image, the inverse (backward) mapping yields a real value (x,y) in the source image. In this case, the pixel gray level g(m,n) at the integral target location (m,n) must be interpolated from the gray levels of the 4 points nearest the point (x,y) shown as an "x" in Figure 6.1. This is called *pixel filling*. In the case of the bottom mapping in Figure 6.1, the pixel at (m,n) in the source image is mapped to a real point (x,y) in the target image shown as "x." This is called *pixel carryover*. The nonintegral point (x,y) is moved to the closest of the integer locations, say, (m_0,n_0) of the target pixel locations, called the *nearest neighbor* of (x,y). The gray level $f(m,n)$ is assigned to that location via $g(m_0,n_0) = f(m,n)$.

As an example of a geometrical mapping, consider

$$a(m,n) = 2.5m + 0.8n, \quad b(m,n) = 0.8m + 2.5n \tag{6.2}$$

The pixel located at (10,10) would become the pixel at

$$(a,b) = ((2.5)(10)+(0.8)(10),\ (0.8)(10)+(2.5)(10)) = (33,33)$$

This mapping is linear and we can see that it maps a point on the diagonal into a point on the diagonal. The mapping is a matrix

$$\begin{vmatrix} a \\ b \end{vmatrix} = \begin{vmatrix} 2.5 & 0.8 \\ 0.8 & 2.5 \end{vmatrix} \begin{vmatrix} m \\ n \end{vmatrix} \tag{6.3}$$

A linear mapping does two things to vectors: i) *contracts or dilates* their length, depending upon the size of the entries and thus of the eigenvalues; and ii) *rotates* the vector. Such transformations are called rigid body motions. Note that for (m,n) = (10,20), which is off the diagonal, that

$$(a,b) = ((2.5)(10)+(0.8)(20),\ (0.8)(10)+(2.5)(20)) = (41,58) \tag{6.4}$$

which is dilated and rotated somewhat (a line from (0,0) through (10,20) would pass through (40,80) but not through (41,58)). A later section considers rotations, translations, scaling, and warping (the latter is nonlinear).

The tools required for geometrical processing are: i) a transformation (mapping) specification; and ii) an algorithm for interpolation, which assigns the gray levels.

Zooming. The process of *zooming* is equivalent to moving the camera closer to (*zooming in*) or farther from (*zooming out*) the scene. Zooming is the transformation obtained by *scaling* each dimension (see the next section): a = a(m,n) = (c_x)m, b = b(m,n) = (c_y)n. Zooming in enlarges the objects in the scene and the scene itself, so that pixels around the image boundary are lost in the new image (c_x, $c_y > 1$). Zooming out reduces the size of the image so that the result is a smaller image of fewer pixels (c_x, $c_y < 1$), but the scene is the same except for the smaller size. Mixed scaling warps, or distorts. For example, $c_x > 1$ and $c_y < 1$ causes the output image to be higher but narrower.

Zooming in allows us to see a part of the image up close, but extra pixels must be inserted. Where do we get the information for these extra pixels? The only information we have, except for some unusual circumstances, is in the existing pixels. The coarsest method of zooming in or out is to respectively repeat pixels or eliminate pixels. However, zooming in with repeated pixels yields a blocky appearance to the resulting image and adds no extra information. Eliminating pixels can lose valuable information.

6.2 Some Useful Geometric Transformations

Translations. A *translation* of a block or region R of an image involves the translation along each of the x-axis and y-axis by respective fixed amounts t_x and t_y. The formulation for the source to target mapping is

$$(x,y) \Rightarrow (r,s), \quad r = x + t_x, \ \ s = y + t_y \tag{6.5}$$

Such a transformation is not actually linear because *(0,0)* maps to (t_x,t_y) rather than mapping to *(0,0)*. Such mappings are called *translations*. They are linear except for the addition of constants. The inverse transformation of the given translation is the target-to-source backward mapping

$$(r,s) \Rightarrow (x,y), \quad x = r - t_x, \ \ y = s - t_y \tag{6.6}$$

Scaling. A transformation that contracts or dilates in each of the dimensions is called a *scaling* mapping. Let c_x and c_y be two scaling constants. The source to target mapping is

$$(x,y) \Rightarrow (r,s), \quad r = c_x x, \ \ s = c_y y \tag{6.7}$$

The target-to-source (inverse or backward) mapping, is

$$(r,s) \Rightarrow (x,y), \quad x = r/c_x, \;\; y = s/c_y \tag{6.8}$$

Scaling is used to zoom in and out. If the scaling constants are greater than unity, then the transformed image is magnified and constitutes zooming in. If, on the other hand, the scaling constants are less than unity, then the image is reduced in size, which is zooming in. As mentioned above, mixed scaling distorts the image (for example, $c_x = 2.2$ and $c_y = 1.6$).

Rotation. The mappings that rotate an image or region are called *rotations*. The forward (source to target) transformations use the Euler equations (x,y) => (r,s)

$$r = x\cos(\Theta) + y\sin(\Theta), \;\; s = -x\sin(\Theta) + y\cos(\Theta) \tag{6.9}$$

The target-to-source backward mapping (r,s) => (x,y) is given by

$$x = (r)\cos(\Theta) - (s)\sin(\Theta), \;\; y = (r)\sin(\Theta) + (s)\cos(\Theta) \tag{6.10}$$

Any matrix transformation may be considered as a rotation in 3-dimensional space of a 2-dimensional object and scaling by dividing the matrix entries by the largest magnitude of the entries and then multiplying the matrix by this magnitude as a scale factor.

Perspective Mappings. A tall building or other object appears to be relatively smaller at the top than at the bottom when the picture is taken from the bottom because the top is farther away from the lens. Objects that rise up toward the top of the image appear to be longer than they should. These are distortions that are linear at any fixed row, but the distortions vary with height. A mapping that can restore the proportions involves using a fixed row contraction scaler c_x and a variable scalar function $c_y(x)$ that varies with x in that it is greater for x near zero and smaller as x moves away from zero (moves downward).

The particular scaler values can be obtained by experimenting. For example, such an undistorting linear transformation may be given by $c_y(x) = sx + r$, where s is the slope and r is the y-intercept, with $c_y(0) = 2$, r=2, etc., and $c_y(M) = 1$. Solving for s and r from these two points, and using, say, $c_x = 0.9$, this becomes

$$a = 0.9m, \;\; b = c_y(x)n = [(-1/M)x + 2]n \tag{6.11}$$

Affine Mappings. Affine transformations (x,y) => (r,s) take the general form

$$\begin{vmatrix} r \\ s \end{vmatrix} = \begin{vmatrix} c_1 & c_2 \\ c_3 & c_4 \end{vmatrix} \begin{vmatrix} x \\ y \end{vmatrix} + \begin{vmatrix} t_x \\ t_y \end{vmatrix} \tag{6.12a}$$

The inverse of this mapping is

$$\begin{vmatrix} x \\ y \end{vmatrix} = \begin{vmatrix} d_1 & d_2 \\ d_3 & d_4 \end{vmatrix} \begin{vmatrix} (r - t_x) \\ (y - t_y) \end{vmatrix} \tag{6.12b}$$

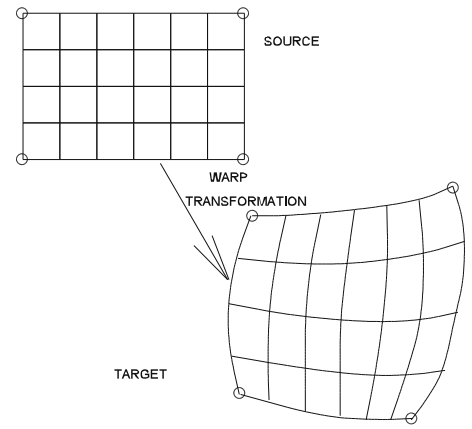
where the matrix $\{d_j\}$ is the inverse matrix of $\{c_k\}$.

The inverse of a linear (matrix) mapping of the form

$$\boldsymbol{M} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} \text{ is}$$

$$\boldsymbol{M}^{-1} = \left(\frac{1}{ad-bc}\right) \begin{vmatrix} d & -b \\ -c & a \end{vmatrix}$$

Figure 6.2. Warping Transformations.



Nonlinear Transformations. Geometric transformations that distort an image are called *warping* mappings (also called transformations). Figure 6.2 shows nonlinear warping. These warp and can be either polynomials or can be approximated by polynomials. For example, a quadratic warping polynomial pair in x and y has the form

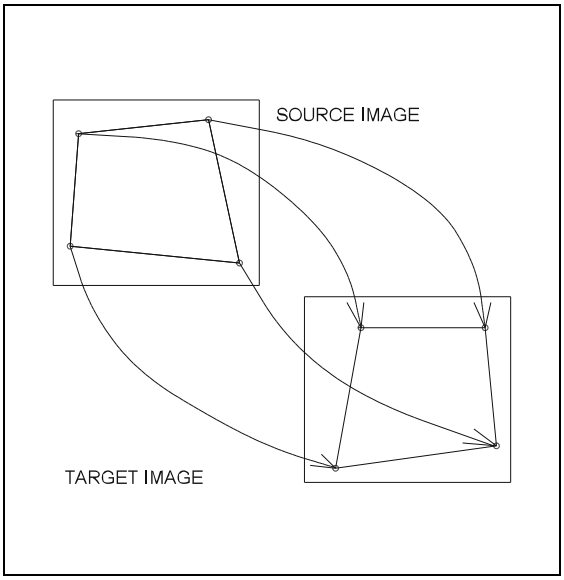
$$r = c_1x^2 + c_2xy + c_3y^2 + c_4x + c_5y + c_6 \qquad (6.13a)$$

$$s = d_1x^2 + d_2xy + d_3y^2 + d_4x + d_5y + d_6 \qquad (6.13b)$$

Warping to remove distortion is a primary objective in many cases where the image is distorted. The nonlinear unwarping transformations can be designed to map the 4 corners of a part of a distorted image into the four corners of an undistorted image, as described in the next section. This is similar to the registration of an image with respect to another image (the undistorted image) discussed in Unit 4..

6.3 Tiepoint Transformations

Figure 6.3. A Tiepoint Transformation.



A *tiepoint* transformation is a nonlinear transformation that maps a region determined by straight lines between 4 source image pixels, called *tiepoints* or *control points*, into a region also determined by lines between the 4 target image tiepoint pixels. Figure 6.3 shows two such quadrilaterals. The transformation is provided by

$$a = a(m,n) = c_1mn + c_2m + c_3n + c_4 \qquad (6.14a)$$

$$b = a(m,n) = c_5mn + c_6m + c_7n + c_8 \qquad (6.14b)$$

There are a total of 8 tiepoints that are to be selected by the user. Upon substituting these 8 points into Equations (6.14a,b), the appropriate 4 into each equation, we obtain 8 equations in the 8 unknowns c_1, c_2,...,c_8, which can be solved by a computer program package. When we have found the coefficients, then we map each pixel (m,n) into a point (x,y) via Equations (6.14).

We may use the nearest neighbor (a,b) in the target image, as the mapped pixel, or we may map each target pixel (a,b) backward (by using the inverse transformation) to obtain a source point and interpolate the gray level from the 4 pixels nearest to it to use as the gray level for the target pixel. The equations for the 4 points are

$$a_1 = c_1m_1n_1 + c_2m_1 + c_3n_1 + c_4, \quad b_1 = c_5m_1n_1 + c_6m_1 + c_7n_1 + c_8 \qquad (6.14c)$$

$$a_2 = c_1m_2n_2 + c_2m_2 + c_3n_2 + c_4, \quad b_2 = c_5m_2n_2 + c_6m_2 + c_7n_2 + c_8 \qquad (6.14d)$$

$$a_3 = c_1 m_3 n_3 + c_2 m_3 + c_3 n_3 + c_4, \quad b_3 = c_5 m_3 n_3 + c_6 m_3 + c_7 n_3 + c_8 \tag{6.14e}$$

$$a_4 = c_1 m_4 n_4 + c_2 m_4 + c_3 n_4 + c_4, \quad b_4 = c_5 m_4 n_4 + c_6 m_4 + c_7 n_4 + c_8 \tag{6.14f}$$

We can use this method to remove distortion by employing an estimated inverse distortion transformation on the source image to obtain a target image essentially free of distortion. The entire image may not be undistortable with a single mapping, but we can map distorted regions via the tiepoint transformation. We may need to experiment by changing the points until it appears to the eye that the distortion is removed. Transformations with a higher degree of nonlinearity can be used (higher degree polynomials), but these require solving a larger set of linear equations in more unknowns and thus use greater computational time. The tradeoff is that a more intricate warping may be achieved.

An important application of the tiepoint transformation is in the registration of one image of a scene with another image of the same scene. The same tiepoints can be selected in each image and the tiepoint transformation can be made as was done in Unit 4. It is important to register two images with respect to each other if the images are to be combined in some fashion.

6.4 Interpolation

The Backward Mapping Technique. In the case of pixel carryover, that is, *forward mapping*, (m,n) maps to a (possibly) nonintegral point (x,y) between the four nearest pixels (see Figure 6.1) and its gray level f(m,n) is assigned to the nearest location of the four corner pixels. Because some pixels may map to points outside of the image and multiple pixels may map (with rounding to integers) to the same pixel location (m_0,n_0), we do not use this method. Instead, we prefer pixel filling via *backward mapping*.

The pixel filling technique constructs the output target image in a pixel-by-pixel manner, one line at a time. In the output image {g(a,b)} to be generated, each pixel location (a,b) is taken in turn, one at a time, and the inverse geometric mapping takes it back to a point (x,y) in the original image {f(m,n)}, where (x,y) may be a nonintegral real number (see Figure 6.1). The point (x,y) must be assigned a gray level f(x,y) based on interpolating the 4 corner points surrounding it (although more than 4 could be used with more computational complexity). Then we put g(a,b) = f(x,y). The interpolation algorithm used affects the quality of the resultant image.

Backward Mapping Nearest Neighbor Interpolation. This is also called *zeroeth order* interpolation because there is no linear, quadratic, cubic, etc., approximation technique involved in the process. The top mapping shown in Figure 6.1 clarifies the process. Each pixel location *(a,b)* in the output image is mapped by the inverse map *(a,b) => (x,y)* into the real point *(x,y)* whose gray level must be found from the 4 corner pixels

$$f(\llbracket x \rrbracket, \llbracket y \rrbracket),\ f(\llbracket x+1 \rrbracket, \llbracket y \rrbracket),\ f(\llbracket x \rrbracket, \llbracket y+1 \rrbracket),\ f(\llbracket x+1 \rrbracket, \llbracket y+1 \rrbracket)$$

where $\llbracket r \rrbracket$ is the greatest integer less than or equal to *r*.

This decision is quick in the case of *nearest neighbor* interpolation, which simply choses the pixel location $(m_0,n_0) = (\llbracket x \rrbracket, \llbracket y \rrbracket)$, $(m_1,n_0) = (\llbracket x+1 \rrbracket, \llbracket y \rrbracket)$, $(m_0,n_1) = (\llbracket x \rrbracket, \llbracket y+1 \rrbracket)$, or $(m_1,n_1) = (\llbracket x+1 \rrbracket, \llbracket y+1 \rrbracket)$ to which (x,y) is closest. It uses the gray level $f(m_j,n_k)$ for g(m,n) that has the nearest location (m_j,n_k), that is

$$g(a,b) = f(m_j,n_k) \tag{6.15a}$$

where (m_j,n_k) satisfies

$$(m_j-x)^2 + (n_k-y)^2 \le min\{(m-x)^2 + (n-y)^2 : m \text{ and } n = \text{nonnegative integers}\} \tag{6.15b}$$

This usually appears good to the eye, but not always. Where the gray levels are changing significantly at higher frequencies, the aliasing causes artificial artifacts in the target image and blockiness.

Bilinear Interpolation. A more satisfactory type of interpolation is to use the 4 corner points to select a point in a plane that passes through the 4 points. Because 3 points in 3-dimensional space determine a plane, and there are 4 points here, a plane does not fit neatly through the 4 points (if a plane is fit through 3 of the 4 points, then the fourth point may be above or below the plane. We use a *bilinear* function that is linear in each of x and y separately, but is nonlinear in both x and y. The form is

$$f(x,y) = c_1xy + c_2x + c_3y + c_4 \tag{6.16}$$

Clearly, if we let y remain fixed, $f_y(x) = f(x,y)$ is linear in x and similarly f(x,y) is linear in y with a fixed x.

If we were to fit a lineal (straight line) function f(x) of a single variable to two points x_1 and $x_2 = x_1 + 1$ along the x-axis from the first value to the second value, we could describe the functional value at every point on the line by

$$f(x_1 + \alpha) = (1-\alpha)f(x_1) + \alpha f(x_2) \tag{6.17}$$

as α goes from $\alpha = 0$ to $\alpha = 1$ ($x_1 + \alpha$ goes from x_1 to x_2).

In a 2-dimensional domain where we consider the function to be linear in one variable for each fixed value of the other variable, we substitute to get

$$f(m+\alpha, n+\beta) = (1-\alpha)(1-\beta)f(m,n) + \alpha(1-\beta)f(m+1,n) + (1-\alpha)\beta f(m,n+1) + \alpha\beta f(m+1,n+1) \tag{6.18}$$

as α and ß go from 0 to 1. This equation provides a quick and easy computation of the gray level at any point

$$(x,y) = \big((1-\alpha)m_1 + \alpha m_2,\ (1-\beta)n_1 + \beta n_2\big) \tag{6.19}$$

in the square determined by the 4 points: (m_1,n_1), (m_1,n_2), (m_2,n_1), and (m_2,n_2).

Fuzzy Interpolation. Consider the case of a transformation from the source to the target image. The inverse transformation takes each pixel location (m,n) in the target back to the real values point (x,y) in the source image via (m,n) = (x,y). The point (x,y) in the source image has 4 nearest pixels that surround it. We need to interpolate the gray level value f(x,y) from the 4 gray levels

$$f_1 = f(\llbracket x \rrbracket, \llbracket y \rrbracket), \qquad f_2 = f(\llbracket x+1 \rrbracket, \llbracket y \rrbracket) \tag{6.20a,b}$$

$$f_3 = f(\llbracket x \rrbracket, \llbracket y+1 \rrbracket), \qquad f_4 = f(\llbracket x+1 \rrbracket, \llbracket y+1 \rrbracket) \tag{6.20c,d}$$

One way to interpolate is to find a typical value that represents f(x,y) from the values f_1, f_2, f_3, and f_4 according to how close the location (x,y) is to their pixel locations. The WFEV (*weighted fuzzy interpolated value*) is such a typical value. Let the 4 source image pixel locations used as arguments in Equation (6.20) be designated by (m_1,n_1), (m_2,n_2), (m_3,n_3), and (m_4,n_4). The WFEV is computed via

$$fg_F = \sum_{(k=1,4)} exp[-\big((m_k-x)^2 + (n_k-y)^2/(2s^2)\big)]\ f_k \ / \sum_{(k=1,4)} exp[-\big((m_k-x)^2 + (n_k-y)^2/(2s^2)\big)] \tag{6.21}$$

This can also be written as

$$f_F = \sum_{(k=1,4)} w_k f_k \tag{6.22}$$

where

$$w_k = exp[-((m_k-x)^2 + (n_k-y)^2/(2\sigma^2))] \ / \sum_{(k=1,4)} exp[-\big((m_k-x)^2 + (n_k-y)^2/(2s^2)\big)] \tag{6.23}$$

$$w_1 + w_2 + w_3 + w_4 = 1 \tag{6.24}$$

Note that pixels close to (x,y) have a greater weighting and pixels farther away have less weighting. The determining factor as to how much relative weight the pixels have is the *standard deviation* σ, which is a spread parameter. A large value for σ means that the Gaussian function is spread out so that pixels farther away will have relatively more weighting. A small value for σ causes pixels farther away to have relatively smaller weighting (more like the nearest neighbor). Without knowing what value to use for σ, we can take σ = 0.6. The midpoint has a distance of $1/\sqrt{2} = 0.707$, so 0.6 appears to be a good value.

We can also use the *Tanimoto* fuzzy set membership function that is faster to compute. The weights are

$$w_k = 1 / \{s((m_k-x)^2 + (n_k-y)^2) + 1\} \tag{6.25}$$

Appendix 6A is a listing of a C/C++ program that performs fuzzy interpolation on PGM grayscale images. Instead of using the 4 pixels in the source image that surround each inversely mapped pixel in the target image, we could use the 9 closest pixels to it. This requires extra computation and sometimes is not worth it. However, if 9 pixels are used then the result is more smoothed and can be sharpened with unmask sharpening.

A General Interpolation Algorithm. The process of transforming and interpolating a source image into a target image using the backward mapping technique is straightforward. We obtain the number of rows and columns in the target image, and then process each taraget pixel consecutively in a row starting with the first row, and proceeding to the next row after that, and so forth, until the last row is processed.

Algorithm 6.1. General Geometric Transformation and Interpolation

```
Mout = No_Output_Rows();                    //Determine no. rows in new image
Nout = No_Output_Cols();                    //Determine no. cols. in new image
for a = 1 to Mout do                        //For each pixel in new image, i.e.,
   for b = 1 to Nout do                     //target image, get locations in original
      x = Inverse_Map_x(a,b);               //image via inverse mapping
      y = Inverse_Map_y(a,b);               //(x,y) is not necessarily pixel location
      x1 = integer(x);                      //Get greatest integer values less than
      y1 = integer(y);                      //or equal to inverse mapped values
      x2 = x1 + 1;                          //Get 4 source pixel locations around
      y2 = y1 + 1;                          //the inverse mapped point (a,b)
      g[a,b] = Interpolate(x1,x2,y1,y2);    //Interpolate source gray levels f(a(j),b(k))
```

The functions *Inverse_Map_x* and *Inverse_Map_y* provide the inverse transformation from (a,b) back to (x,y) which may not be located at a pixel (m,n) (may not have integral components). We then find the 4 pixel points surrounding (x,y), which are (x_1,y_1), (x_1,y_2), (x_2,y_1), (x_2,y_2). The function *Interpolate( )* applies interpolation using the source gray levels $f(x_1,y_1)$, $f(x_1,y_2)$, $f(x_2,y_1)$, $f(x_2,y_2)$ to obtain the target gray level g(a,b). More source pixels may be used, but at greater computational cost and extra smoothing.

The functions *No_Output_Rows* and *No_Output_Cols* map the 4 corners in the original image into the 4 corners of the target image and find the minimum and maximum of these. The corner with the minimal x value is taken to be 0 plus a translation T_x while the minimal y value is taken to be 0 plus a translation T_y. If any of the interpolation target points are outside of the mapped original image, we put g(m,n) = 0 (background). Appendix 6A provides a program listing in C that does affine and other transformations on PGM P5 images and zooming using fuzzy interpolation with the Tanimoto fuzzy set membership function.

6.5 Computer Experiments

***Enlargement with* Matlab**. Figures 6.4 and 6.5 show the source (original) images *lena256.tif* and *shuttle.tif*. We zoomed in on each of these using bilinear interpolation. Figure 6.6 displays *lena256.pgm* with magnification of 1.8 while Figure 6.7 shows the 1.8 magnification of *shuttle.pgm*.

Figure 6.4. Original Lena.

Figure 6.5 Original shuttle.

Fig. 6.6. Bilinear 1.8 zooming of Lena.

The enlarged images look quite good with the bilinear interpolation that is linear separately in each of x and y, although it has an xy term. Bicubic interpolation is similar but instead of a linear function in each of x and y separately, it uses a cubic function in each of x and y with cross products. It is the best theoretically, but takes more computation and time, and often appears no better to the human eye than bilinear.

Figure 6.7. Bilinear 1.8 zooming of shuttle.

Figure 6.8. Nearest neighbor 2x2 Lena.

Figure 6.9. Bilinear 2x2 Lena.

Matlab *Experiments with Zooming*. To get started, we click on the Matlab icon (MS Windows) or type in matlab at the command prompt (UNIX). When it comes up we type the following in the command window.

```
>> I1 = imread('lena256.tif');              %read in image
>> imshow(I1);                    %show image
>> I2 = imresize(I1, 1.8,'nearest');        %resize image: 1.8x1.8 times area, nearest neighbor
>> figure, imshow(I2);                      %show nearest neighbor enlarged image
>> I3 = imresize(I1, 1.8, 'bilinear');      %resize image 1.8x1.8, bilinear interplolation
>> figure, imshow(I3);                      %show bilinear interpolated image
>> I4 = imresize(I1,1.8,'bicubic');         %resize image 1.8c1.8, bicubic interpolation
>> figure, imshow(I4);                      %show bicubic interpolated image
```

The nearest neighbor sometimes looks good, especially if it is smoothed slight, but it can sometimes look blocky. Here the result is good. Figure 6.8 shows the 1.8x1.8 nearest neighbor extrapolated Lena. Figures 6.9 and 6.10 show the respective use of bilinear and bicubic interpolation. While bilinear looks good in this case, the bicubic does not look as good due to the higher degree polynomial. Figure 6.11 shows the results of the fuzzy interpolation.

Figure 6.10. Bicubic 1.8 Lena.

Figure 6.11. Fuzzy interpolation 1.8 Lena.

6.6 Affine Transformations in C

Using Matlab for Affine Transformations. We use the function maketform() to make a transformation that is really a 2x2 matrix, but the function requires that we put in the 3x3 scaled Euler transformation. To do this we use *0* to complete each of the first two rows with a third column element and then use *0 0 1* as the third row, where the 1 in the third column places the 2-D plane in 3 dimensions. Thus if we want to use the 2x2 matrix given below on the left, we actually use the second 3x3 matrix on the right.

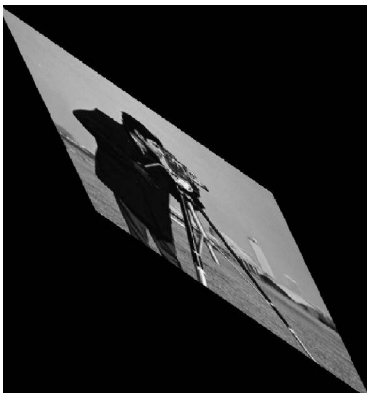
$$\begin{vmatrix} 1.4 & 1.1 \\ 0.5 & 1.2 \end{vmatrix} \qquad \begin{vmatrix} 1.4 & 1.1 & 0 \\ 0.5 & 1.2 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

The Matlab commands to obtain the images shown below are

```
>> I = imread('cameraman.tif');                                  //read in cameraman image
>> mytform = maketform('affine',[1.4 1.1 0; 0.5 1.2 0; 0 0 1]);  //make matrix transformation
>> J = imtransform(I,mytform);                                   //transform the input image
>> imshow(I), figure, imshow(J);                                 //show input & output images
```

Figure 6.12. Original cameraman image.

Figure 6.13. Transformed Cameraman image.



Fuzzy Interpolation for Affine Transformationsl. Here we use our C/C++ program provided in Appendix 6A that implements the Gaussian fuzzy set membership function to obtain the weights for the interpolation (see the *interpolate()* function in the program). Upon compiling and running the program, it will ask the user to type in the names of the input and output PGM files and enter the spread parameter σ (the values 0.4 to 0.6 are quite good). We input the file *shuttle.pgm*. The user is asked to choose between *zoom, affine, rotate* and *warp*. Here, we select affine.

The user is then asked to input the first row of two values so we input *1.4 0.8 <ENTER>*. The user is next asked for the second row, so we input the two values *1.2 1.0 <ENTER>*. The inverse 2x2 matrix is then printed on the screen and also the matrix that was entered. The user is asked to accept the matrix or redo it, in which case the matrix input process is repeated. Upon running the program on the shuttle image as the input, we obtain the results shown below in Figure 6.15.

Figure 6.14. The input shuttle image.

Figure 6.15 The affine shuttle image result.

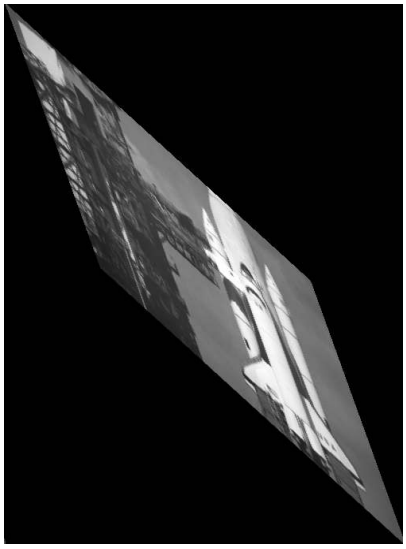


Figure 6.16. Transformed building.

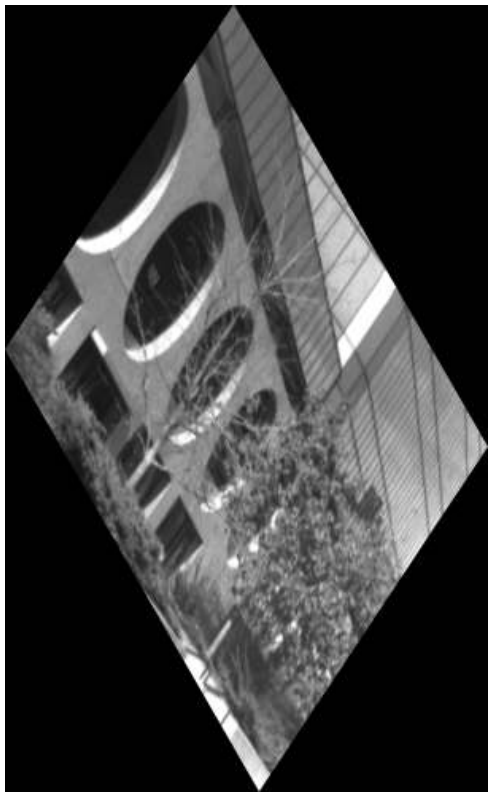


Figure 6.16 uses the affine transformation shown below.

$$\begin{bmatrix} 0.5 & 0.86 & 0 \\ -0.6 & 0.9 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 17. Original building.

6.7 Exercises

6.1. The 2x2 matrix

$$\begin{vmatrix} c_1 & -c_2 \\ c_2 & c_1 \end{vmatrix}$$

maps a vector (m,n) into a vector (a,b) linearly. Show that this matrix can be considered as a combination of a scaling and a rotation whenever $c_3 = -c_2$. Find the angle Θ of rotation by dividing by $(c_1^2 + c_2^2)^{1/2}$ and converting the resulting entries to sines and cosines. Use the scaler outside of the matrix.

6.2. Write an algorithm for user-given tiepoints to warp an image (4 control points inr each of the source and target images).

6.3. Write a C/C++ computer program to zoom in or zoom out by a factor given by the user. Permit the vertical and horizontal factors to be different so the aspect angle (ratio of height to width) to be changed. The main item here is to obtain the output image of the new set of pixels and then for each of these pixels to find a gray level from the original pixels by backward mapping and interpolation.

6.4 Modify the program in problem 6.3 above so that rotations can be done.

6.5. Compute the scaling function that contracts the first column by a factor of 0.5 and expands the last column by a factor of 1.5 such that all columns in between the first and last are scaled a proportion of the way between 0.5 and 1.5.

6.6. Describe in detail an algorithm that performs linear geometrical transformations (write the high level pseudo-code for the algorithm).

6.7. Write out an algorithm for general affine transformations that use bilinear interpolation.

6.8. Write a C/C++ computer program that implements the algorithm of Exercise 6.7.

6.9. Write an algorithm that finds the inverse mapping of any affine transformation.

6.10 Use Matlab to read in shuttle.tif as I1, convert it to black and white to obtain Ibw, then erode Ibw to get Iout1 and dilate Ibw to get Iout2. Now subtract Iout2 from Iout1 to get the result as Iout. Now thin this image to get trimmed lines. Show all results.

Solution: Matlab does not allow the subtraction of a black and white image from a black and white image because both have logical data types for the pixel values. The function imsubtract() must operate on data types uint8 (8-bit unsigned integers for grayscale) or double (floating point). So we have to work around this. From the command line, we type the following.

```
>> I1 = imread('shuttle.tif');          //read in image
>> imshow(I1);                          //show image
>> Ibw = im2bw(I1, 0.75);               //convert to black and white with im2bw();
>> imshow(Ibw);                         //show b&w image
>> Iout1 = bwmorph(Ibw, 'dilate');      //dilate b&w image
>> Iout2 = bwmorph(Ibw, 'erode');       //erode b&w image
>> Iout = Iout1 - Iout2;                //subtract eroded image from dilated image
>> figure, imshow(Iout);                //show subtracted image
>> Ioutthin = bwmorph(Iout, 'thin');    //thin subtracted image with bwmorph();
>> figure, imshow(Ioutthin);            //show the thinned image
```

Figure 6.16. Original image.

Figure 6.17. Black & white image.

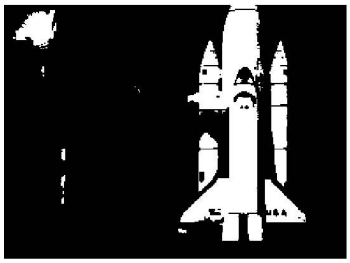


Figure 6.18. Subtracted image.

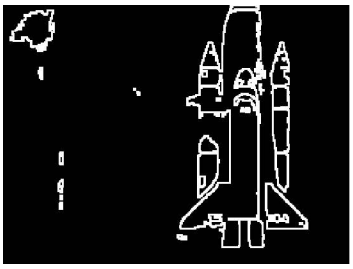
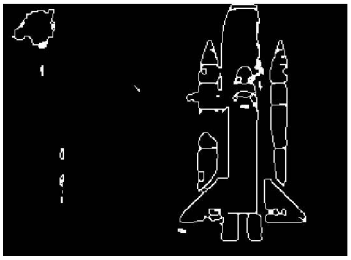


Figure 6.19. Thinned image.



6.11 Write an algorithm that solves for the coefficients of the polynomial mapping that maps the region within 4 tie-points into another region within the mapped 4 tiepoints.

6.12 Compile the programs *fzntrp.c*, and run it to zoom in on the image *lena256.pgm*. Show the before and after images.

6.13 Zoom by the factors $c_x = 1.4$ and $c_y = 1.0$ on the program *shuttle.pgm*.

6.14 What can one say about the edges in a reduced image? In an enlarged image?

6.15 Develop an algorithm that solves for the coefficients in Equations (6.14c-f).

6.16 Write a C/C++ function that solves 8 equations in 8 unknowns for the algorithm in 6.14 above.